Certificación PCAP

Lección 5: Programación Orientada a Objetos

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

1. Objetivos 2
2. Marco teórico 3
2.1. Clases y objetos 3
3. Programación orientada a objetos en python 8
3.1. Definir una clase 8
3.2. Definiendo atributos en una clase. Inicializando objetos 10
3.3. Definiendo métodos en las clases 14
3.4. Variables de instancia vs variables de clase 17
4. Herencia 22
4.1. Herencia única 22
4.1.1. Definición de una relación superclase-subclase en Python 23
4.1.2. Variables de clase en la herencia 23
4.1.3. Herencia de métodos 24
4.1.4. Variables de instancia en la herencia 25
4.1.5. Conflictos en herencia única 30
4.2. Herencia múltiple 32
4.2.1. Definición de una clase con herencia múltiple 33
4.2.2. Herencia múltiple. Variables de clase, atributos y métodos sin conflicto 34
4.2.3. Herencia múltiple. Resolución de conflictos 36
4.3. Conclusión sobre la herencia 39
5. Puntos Clave 40

Lección 5: Programación orientada a objetos

1. Objetivos

En esta lección nos centraremos en el estudio de la programación orientada a objetos y sus principales características.

La primera parte de la lección tendrá un enfoque un poco más teórico donde repasaremos los conceptos clave de los objetos, clases y la herencia, para posteriormente, en la segunda parte de la lección pasar a ver estos conceptos de una manera práctica, haciendo especial énfasis en aquel tipo de preguntas que podemos encontrar con mayor probabilidad al realizar el examen PCAP. Finalmente terminaremos la lección, centrándonos en lo relativo a la herencia ya que suele ser un tema bastante recurrente en el PCAP.

2. Marco teórico

Posiblemente nos suene que, durante el desarrollo de esta asignatura, hayamos mencionado en algunas ocasiones algo similar a “en Python todo son objetos”, pero, ¿qué significa esto? ¿Tiene alguna implicación importante más allá de ser una frase cliché que escuchamos en el desarrollo de cualquier cursillo de Python con el que nos topemos? Pues, a decir verdad, y cómo seguramente se podrá imaginar, la programación orientada a objetos es un aspecto clave en el mundo del desarrollo hoy en día, así como también lo es en la certificación PCAP, ya que podemos estar absolutamente seguros de que encontraremos más de una pregunta cuyo contenido tenga que ver con el trabajo con clases y objetos. Según vimos en la primera lección, alrededor de un 34% del peso del examen recae sobre este punto, lo cual nos da una pista de la importancia de su estudio.

Para comenzar con el estudio de esta lección, hablaremos primeramente sobre los conceptos fundamentales de la programación orientada a objetos. Dichos conceptos son, quizá, un poco más complejos que los vistos hasta ahora en la asignatura, pero una vez comprendidos y asimilados los mismos es posible ver la inequívoca lógica que rige la manera de trabajar con ellos en Python.

Empecemos, pues, por el principio. Las clases y los objetos.

2.1. Clases y objetos

Uno de los primeros conceptos de que debemos definir y entender son las “clases”, ya que son las “originarias” de los objetos.

¿Qué es una clase?

Para poder explicar el concepto de clase, normalmente, se suele hacer una analogía con el mundo “no programático” es decir, una clase es un concepto que podemos percibir y entender mucho más claramente al utilizar ejemplos de la vida cotidiana, y posteriormente pasar a su aplicación a nivel de código, y más particularmente a la manera en la que se codifica en Python. Sigamos pues este patrón.

Cuando se trata de definir algo, habitualmente suele ser una muy buena idea recurrir primeramente a un diccionario, y obviamente no a uno de Python, sino al de la Real Academia Española, veamos lo que dice acerca de la definición del término “clase”.

Clase. *1. f. Conjunto de elementos con caracteres comunes.*

No tenemos que buscar mucho, ya que en la primera acepción encontramos la definición que posiblemente sea la más acertada y relacionada con la idea de clase que buscamos.

Como bien podemos leer, una “clase” no es más que un conjunto de elementos que tienen determinadas similitudes entre sí. Esta definición nos da pie a ahondar más, ya que si esto es así podemos afirmar sin lugar a duda que **todo** pertenece a una clase, es decir, cualquier cosa que encontremos en nuestro día a día, sea **objeto** (y aquí comienza a aparecer el término) físico o idea intelectual la podemos clasificar y agrupar en distintas clases o familias.

Por poner algunos ejemplos, por las mañanas nos vestimos con elementos de la familia “ropa”, desayunamos algo que pertenece a la familia de “alimentos”, vamos al trabajo (si tenemos que desplazarnos) en **objetos de la clase** “medios de transporte”. Estos últimos son, posiblemente, el ejemplo por excelencia utilizado por la gran mayoría de cursos para explicar la programación orientada a objetos y el concepto detrás de esto.

Utilizar los ejemplos enumerados anteriormente (ropa, alimentos y transporte) son muy útiles para poder explicar la idea detrás del término clase.

Cuando hablamos de “ropa” no hacemos referencia a ninguna prenda en particular, tampoco lo hacemos al pensar en “alimentos” o “medios de transporte” y es aquí donde está la clave para entender el concepto de clase. Las clases no son nada “particular” o que apuntan a una “entidad” u “**objeto**” concreto, la clase viene muy bien representada por los términos generales “ropa”, “alimentos” y “medios de transporte”, los cuales hacen referencia a una serie de cualidades o características comunes que tienen cada una de las entidades particulares que pueden formar parte de una clase.

¿Qué es un objeto?

Una vez hemos asimilado este concepto en relación a las clases, es muy intuitivo asimilar igualmente el concepto de **objeto**, ya que ambos vienen prácticamente de la mano. Mientras que una clase es una "generalización" abstracta de las características que han de tener los integrantes de dicha clase, los objetos son las entidades **concretas** que cumplen con esa serie de características y que por tanto forman parte de esa familia. Son, en resumidas cuentas, las "encarnaciones" (o como se dice normalmente en programación, **instancias**) de las clases.

Volviendo a los ejemplos de antes (ropa, alimento, medios de transporte) si quisiéramos identificar algunos objetos de estas clases podríamos mencionar por ejemplo la camiseta que llevamos puesta ahora mismo, la tostada que quizás hemos desayunado esta mañana o nuestro coche, el cual hemos utilizado para desplazarnos al trabajo o universidad.

Estos ejemplos traen a escena otro de los aspectos fundamentales de las clases, y es la posibilidad de **crear subclases**, entendiendo estas como sub-familias o subgrupos que se puedan formar dentro de una clase existente.

Centrémonos en el ejemplo de la clase "alimentos". Al hablar del término "tostada" podríamos pensar que estamos refiriéndonos a un objeto, sin embargo, dicho término puede representar a la tostada que desayunamos ayer, o la de hoy, o la de mañana, no representa a una entidad única y exclusiva, con lo cual, llegaríamos a la conclusión de que "tostada" forma una nueva clase, concretamente, una subclase de la clase padre (o **superclase**) "alimentos", mucho más específica, eso sí, pero una nueva clase, al fin y al cabo. Igualmente, podemos aplicarlo al resto de ejemplos, donde "camiseta", "pantalón" o "sudadera" son ejemplos de posibles subclases de la clase "ropa" o "coche", "moto" y "avión" lo son de la clase "medios de transporte".

Observaremos que las subclases tienen una relación muy estrecha con la superclase de la que provienen, ya que como es lógico, al provenir de esta superclase "heredarán" de la misma determinadas cualidades y características. Esta cualidad se denomina **herencia** y es un pilar clave en la programación orientada a objetos. Veremos más adelante como trabaja Python con la herencia y lo importante que resulta en las preguntas del examen.

En el caso de los medios de transporte, por ejemplo, la subclase “coche” hereda de la superclase “medios de transporte” la función para la cual se utiliza, transportar personas y/o mercancías y a la vez es una subclase mucho más específica.

A continuación, podemos ver un pequeño esquema representativo de la relación entre la superclase “medios de transporte” y distintos niveles de subclases, comenzando desde una clasificación más general según el medio por el cual se muevan (terrestres, marítimos o aéreos) y las subclases que derivan de cada una de estas.

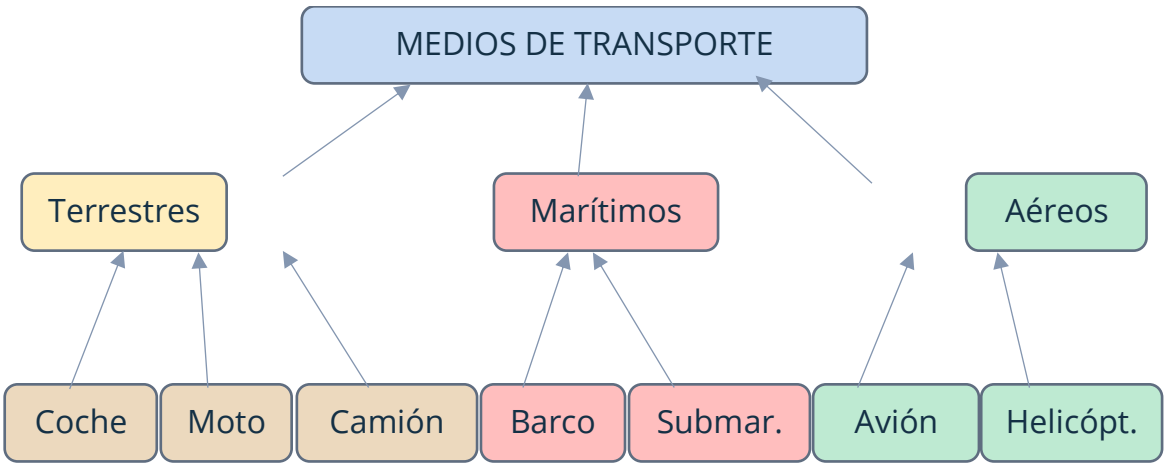

Figura 2.1. Esquema de la relación entre clases y subclases

Una última consideración en relación a la herencia. En Python, esta herencia puede ser múltiple. ¿Qué quiere decir esto? Pues bien, utilicemos en este caso las clases del esquema que vemos a continuación.

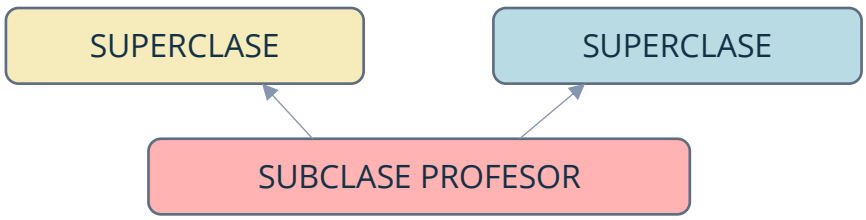

Figura 2.2. Representación gráfica de la herencia múltiple en las subclases

Como vemos, tenemos dos superclases de partida; la clase “profesor” y la clase “investigador” y una subclase “profesor investigador”. Pues bien, este es un caso de herencia múltiple, donde tenemos dos (o más) superclases de las cuales depende la subclase en cuestión.

Esta subclase heredará características y cualidades tanto de la superclase “profesor” como de la clase “investigador”.

Como se puede imaginar, esta posibilidad de la herencia múltiple dota de mucha versatilidad al paradigma de la programación orientada a objetos, ya que permite crear subclases enriquecidas con características de otras clases ya existentes, lo cual permite una mayor reutilización de código, lo que se traduce en rapidez de desarrollo y limpieza del mismo. Sin embargo, no todo son ventajas, ya que, al existir la posibilidad de una herencia múltiple, aparece también la posibilidad de que ocurran “conflictos” entre las características que la subclase hereda de las superclases.

Este escenario es uno de los principales en el que nos podremos encontrar en el examen del PCAP por lo cual será muy importante entender las reglas que sigue Python para “resolver” estos aparentes conflictos de la herencia múltiple.

El tópico de la programación orientada a objetos puede ser tan extenso como se quiera, pero puesto que en esta asignatura tenemos como objetivo principal enfocarnos en los contenidos concretos de la certificación PCAP, estableceremos aquí el límite del contenido puramente teórico para pasar a ver de una manera más práctica como todos estos elementos toman forma en Python.

3. PROGRAMACIÓN ORIENTADA A OBJETOS EN PYTHON

Una vez que hemos entendido la idea conceptual del paradigma de la programación orientada a objetos, es momento de pasar a lo concreto, es decir, cómo se traducen todas estas ideas teóricas a la codificación en Python, y cuáles son las reglas que rigen la definición de clases y objetos y el trabajo con los mismos.

3.1. Definir una clase

Hablemos de lo más básico, ¿cómo podemos definir una nueva clase en Python? A continuación, vemos un ejemplo de lo que podría ser considerado como la clase más "sencilla" que podemos crear en Python.

```
1.      >>> class MiPrimeraClase:
2.      >>>     pass
3.
```

La sintaxis para definir una nueva clase utiliza el término reservado "class" (minúscula) seguido del nombre de la nueva clase y los dos puntos ":".

Está claro por qué esta es la clase más sencilla que podemos definir, ya que se observa claramente que no tiene ningún contenido, simplemente se utiliza la partícula "pass", otro término reservado de Python, para que la creación de la clase no de errores. De la misma manera podríamos empezar a incluir algún tipo de contenido, lo más sencillo, mostrar un mensaje al crear la clase.

```
1.      >>> class MiPrimeraClase:
2.      >>>     print("Hola mundo, esta es mi primera clase")
3.
4.      Hola mundo, esta es mi primera clase
5.
```

Ahora, una vez hemos definido y creado nuestra nueva clase, podemos utilizarla para crear objetos nuevos que pertenezcan a dicha clase. Este procedimiento mediante el cual se crea un nuevo objeto utilizando una clase se denomina **instanciación**, y la manera de codificarlo en Python es la siguiente.

```
1.      >>> objeto_1 = MiPrimeraClase()
2.
```

Como vemos, lo primero es decidir el nombre de variable al que asignaremos el nuevo objeto que crearemos, en nuestro caso, haciendo un alarde de creatividad, le llamaremos "objeto_1", posteriormente utilizamos el operador de asignación "=" y el nombre de la clase seguido de los paréntesis. En el caso de necesitar de parámetros de entrada para definir el nuevo objeto, tales valores irían entre estos paréntesis. Puesto que la clase de ejemplo que hemos definido es tan simple que no necesita argumentos de entrada, en nuestro caso no los hay.

Una vez llegados a este punto, podemos preguntarnos qué podemos añadir a las clases para que los objetos instanciados posean dichas características, ya que la clase de ejemplo vista anteriormente carece de utilidad más allá de los límites didácticos. Pues bien, generalmente, podemos decir que las características/atributos de las que podemos dotar a las clases y consecuentemente a sus instancias/objetos se dividen en 3 grupos:

- Primeramente, normalmente los objetos tienen un nombre que los identifica.
- Los objetos tienen un conjunto de propiedades individuales que los identifican como entidades únicas. A estas propiedades se le suele denominar **atributos**.
- Finalmente, los objetos pueden tener una serie de "habilidades" con las que pueden alterarse a sí mismos o a otros objetos. Estas habilidades reciben habitualmente el nombre de **métodos**.

Se suele decir, a modo de resumen, que cada objeto puede tener un "sustantivo" que hace referencia a un nombre que los identifica, uno o más "adjetivos", valores que los describen y finalmente, uno o varios "verbos" que representan la "acción o capacidad" de hacer alguna tarea o modificación.

Para entenderlo mejor, utilicemos otro de los famosos ejemplos recurrentes en la explicación de estos conceptos.

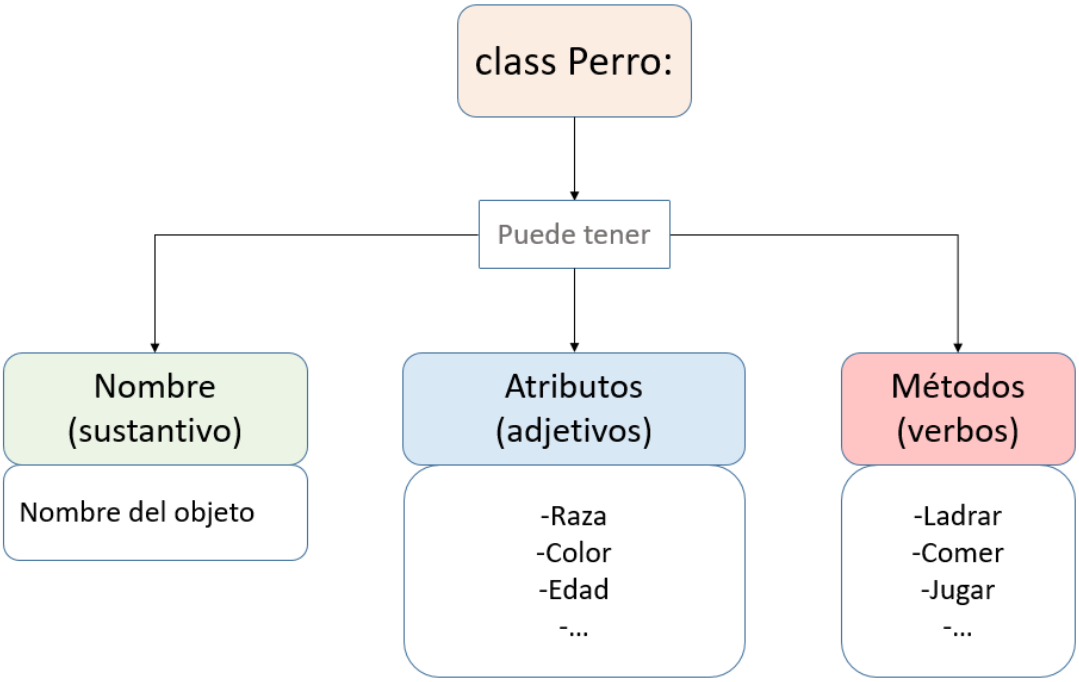

Figura 3.1. Representación esquemática de las propiedades de una clase

El esquema anterior, representa estos tres grupos de características de las que podemos dotar a los objetos que construyamos utilizando esta clase. Ahora bien, vemos que al definir la clase

3.2. Definiendo atributos en una clase. Inicializando objetos

Veamos ahora la manera en la que se definen los atributos de una clase en Python.

Supongamos que queremos crear una clase tal que al instanciar un objeto este tenga de inicio una serie de atributos que le definan unívocamente.

Por ejemplo, basándonos en el ejemplo de la clase perro, que, al instanciar un nuevo objeto de la clase perro, queremos que este tenga un nombre, por supuesto, una raza y un color.

¿Cómo habría que crear la clase e instanciar el objeto para que esto ocurriese?

Pues bien, para llevar a cabo esta "inicialización" de los objetos, Python hace uso de la función "__init__", también llamada **constructor**.

Veamos a continuación su modo de empleo.

```
1.      >>> class Perro():
2.      >>>     def __init__(self, raza, color):
3.      >>>         self.raza = raza
4.      >>>         self.color = color
5.
6.      >>> mi_perro = Perro("labrador","chocolate")
7.
8.      >>> print(mi_perro.raza, mi_perro.color, sep= " || ")
9.      labrador || chocolate
10.
```

Observemos en detenimiento la definición de la clase y la creación del objeto.

Primeramente, en la línea 1 encontramos la definición de la clase, donde hemos particularizado el nombre de la misma para nuestro ejemplo.

En la línea 2 hacemos uso de la función __init__ para inicializar los valores de los atributos que tendrá el objeto justo después de su creación. Esta función siempre necesita al menos de un argumento de entrada, el "**self**". Este "self" representa al objeto que se crea a la hora de instanciar la clase, es decir, cada vez que se cree un nuevo objeto de la clase "Perro", dicho objeto vendrá representado dentro del código de la clase por la palabra "self". No debemos olvidar que **la función constructora siempre espera al menos este argumento**.

Esta función bien puede recibir más argumentos, tal y como vemos en el ejemplo. A lo hora de definir la función podemos requerir que se introduzca el número de parámetros que queramos, en nuestro caso, hemos requerido que se introduzcan 2 argumentos extra a la hora de instanciar la clase: una raza y un color.

En las líneas 3 y 4 definimos esos atributos haciendo uso de la nomenclatura del punto. Cualquier atributo que deseemos inicializar lo haremos de esta forma "**self.nombreAtributo = valor**". En este caso, vemos que el valor de estos atributos "raza" y "color" viene definido por los argumentos pasados en la creación del objeto en la línea 7 "labrador" y "chocolate", sin embargo, pueden tomar cualquier valor, *i.e.* una lista, una tupla, una cadena, etc.

Por otro lado, en la línea 7 encontramos la creación del objeto (la instancia de la clase), la manera de hacerlo es la siguiente:

mi_nuevo_objeto = miClase(argumento1, argumento2,..., argumentoN)

Donde cada variable entre paréntesis hará referencia a cada uno de los argumentos necesitados por la función constructora.

En la línea 8, para finalizar, observamos que se utiliza también la nomenclatura del punto para acceder al valor de los atributos que tiene el objeto.

Pero, supongamos ahora que no nos interesa que cualquier usuario tenga la posibilidad de acceder al contenido de estos valores, ya sea porque contienen información delicada, o simplemente porque queremos evitar que el usuario final, por mero desconocimiento o error, altere de alguna manera la información almacenada y genere algún tipo de problema.

La manera de crear variables ocultas en Python es muy sencilla; simplemente tenemos que añadir dos guiones bajos "__" antes del nombre de los atributos. Veamos esta posibilidad en otro ejemplo.

```
1.      >>> class CuentaUsuario:
2.      >>>     def __init__(self,usuario,password):
3.      >>>         self.usuario = usuario
4.      >>>         self.__password = password # Definimos el
5.                                             # atributo privado
6.
7.      >>> mi_cuenta = CuentaUsuario("Fernando","1234abdc")
8.
9.      >>> print(mi_cuenta.usuario)
10.     >>> print(mi_cuenta.__password)
11.
12.     Fernando
13.     Traceback (most recent call last):
14.     AttributeError: 'cuentaUsuario' object has no
15.        attribute '__password'
16.
```

Vemos claramente cómo mientras no tenemos ningún tipo de problema para acceder e imprimir por pantalla el atributo **público** "usuario" intentar acceder a "password", que sí lo es, devuelve un error que nos notifica que dicho atributo es inexistente, o al menos para el usuario así lo parece.

De esta manera es como tiene lugar la llamada **encapsulación** en Python, donde cualquier elemento definido dentro de la clase que vaya precedido de dos guiones bajos **solo será accesible desde dentro de la misma clase.**

Indicar que hay una forma de acceder al valor de un atributo encapsulado directamente ya que Python no lo convierte en un atributo totalmente privado, sino que lo renombra a:

_nombreClase__nombreAttributo

Por lo que utilizando este nuevo nombre se podría acceder a su valor u modificarlo.

```
>>> class CuentaUsuario:
>>>     def __init__(self,usuario,password):
>>>         self.usuario = usuario
>>>         self.__password = password # Definimos el
                                       # atributo privado

>>> mi_cuenta = CuentaUsuario("Fernando","1234abdc")

>>> print(mi_cuenta._CuentaUsuario__password)
1234abdc
```

Recalcar que, aunque existe esta forma de acceder a este tipo de atributos y hay que conocerla de cara al examen de certificación, no se debe usar a la hora de programar ya que sería una mala práctica.

Resumimos las principales características de la función constructora __init__ en los siguientes puntos:

- La función __init__ siempre tiene este nombre, no puede cambiarse, es un convenio de Python.
- Si no define la función __init__ python utilizará por defecto un constructor vacío.
- Esta función debe al menos siempre tener un parámetro, el ***self***. Este parámetro por convenio se llama self, pero se podría utilizar otro nombre.
- Esta función no debe devolver nada. Si intentamos devolver un valor nos arrojará un error. La única excepción permitida sería devolver None.

```
>>> class miClase:
>>>     def __init__(self):
>>>         return 3

>>> objeto = miClase()
Traceback (most recent call last):
TypeError: __init__() should return None, not int

```

> La función __init__ siempre se ejecuta "automáticamente" (implícitamente) al crear cada objeto, de ahí su nombre y que se utilice para inicializar el estado de los objetos. Un ejemplo simple bastará.

```
>>> class miClase:
>>>     def __init__(self):
>>>         print("Este print se ejecuta
                automáticamente al instanciar el objeto")

>>> objeto = miClase()
Este print se ejecuta automáticamente al instanciar
el objeto

```

Este aspecto de la función __init__ suele ser bastante recurrente en las preguntas del examen, con lo cual debemos intentar recordar esta característica.

3.3. Definiendo métodos en las clases

En la sección anterior hablamos sobre los atributos (adjetivos) que puede contener una clase y los objetos que pertenezcan a ella, sin embargo, ahora ese momento de hablar de cómo dotar a las clases y objetos con la capacidad de realizar alguna acción que pueda modificar el estado de otros objetos o de ellos mismos. Estas acciones reciben el nombre de **métodos**, y la manera en la que se codifican es la siguiente

```
1.      >>> class Perro():
2.      >>>     def __init__(self, raza, color):
3.      >>>         self.raza = raza
4.      >>>         self.color = color
5.      >>>
6.      >>>     def ladrar(self):
7.      >>>         print("guau")
8.
9.
10.     >>> miPerro = Perro("labrador", "chocolate")
11.     >>> miPerro.ladrar()
12.     guau
13.
```

En el ejemplo anterior hemos reutilizado la clase "Perro", sin embargo, ahora centraremos nuestra atención en la definición del método "ladrar" que hay a continuación del constructor.

Podemos observar que la sintaxis es muy similar a la utilizada en el constructor, de hecho, el constructor es un caso especial de método. Al fin y al cabo, estos métodos no son más que funciones dentro de clases, incluso vemos como se utiliza la palabra reservada "def" para definirlos.

No resulta extraño, por lo tanto, que los métodos necesiten también al menos el argumento de entrada "self" que hace referencia al propio objeto instanciado y que además se pueda requerir la introducción de argumentos extra para realizar la acción de la cual se encargue el método. En este ejemplo en particular, vemos en la línea 6 que el método "ladrar" no requiere estos argumentos extra ya que entendemos que el ladrido de un perro, o, mejor dicho, la onomatopeya, será la misma sea cual sea el perro. Es por ello que cualquier objeto nuevo que definamos, al invocar al método ladrar, devolverá por pantalla la cadena "guau". Cabe destacar que, al igual que con los atributos, la manera de invocar a los métodos de los objetos se lleva a cabo a través de la nomenclatura del punto, es decir, para invocar a estos métodos la sintaxis genérica para hacerlo es:

"**nombre_objeto.nombre_metodo(parámetro1, parámetro2,...,parámetroN)**"

Puesto que los métodos son al fin y al cabo funciones, todo lo visto hasta ahora con respeto a ellas es aplicable aquí también.

Tenemos, por un lado, la posibilidad de pasar argumentos de entrada a los métodos de las 3 maneras vistas en la lección de las funciones. Primeramente, lo más normal es pasar los argumentos de manera posicional, es decir, teniendo en cuenta el orden en el que están definidos en el método. También podemos hacerlo por nombres. Y finalmente, podemos hacerlo de manera mixta, siempre respetando la regla de que los argumentos posicionales siempre se pasan primero.

Podemos utilizar "return" para devolver un valor al invocar al método. De la misma manera que en las funciones, invocar a un método puede devolver como resultado algún tipo de variable, esta devolución de datos se lleva a cabo mediante la sentencia "**return variable**".

Lo vemos en las siguientes líneas de ejemplo, donde hemos implementado una clase llamada "Calculadora" cuyo único método es el de restar los parámetros que se introduzcan en la invocación al método. No es una calculadora muy versátil, desde luego, pero cumple con su propósito didáctico.

```
1.      >>> class Calculadora:
2.      >>>     def resta(self, num1, num2):
3.      >>>         return num1 - num2
4.
5.      >>> cuenta = Calculadora()
6.      >>> res1 = cuenta.resta(num2=5, num1=2)
7.      >>> res2 = cuenta.resta(5, 2)
8.      >>> print(res1, res2, sep=" || ")
9.      -3 || 3
10.
```

Observamos claramente cómo es posible pasar los argumentos de entrada de manera posicional y por nombre de variable y cómo afecta esto al resultado final.

Otro aspecto importante que es necesario mencionar con respecto a los métodos, es la posibilidad que también ofrece Python de que estos puedan declararse como privados. La manera de hacerlo es la misma que con los atributos, añadiendo dos guiones bajos antes del nombre del método.

```
>>> class MiClase():
>>>     def __metodoprivado(self,arg):
>>>         return arg * 2

>>> obj = MiClase()
>>> res = obj.__metodoprivado(2)
Traceback
AttributeError: 'MiClase' object has no
attribute '__metodoprivado'

```

Una vez visto cómo se definen los métodos en las clases, podemos comenzar a vislumbrar el increíble potencial que tiene el paradigma de la programación orientada a objetos, donde, al contrario que en el enfoque procedimental, los datos, variables y funciones se "fusionan" en los objetos otorgando a estos una mayor flexibilidad y versatilidad muy aprovechables a la hora de desarrollar.

Ahora tenemos la capacidad de crear múltiples clases y enriquecer estas con todo tipo de atributos y métodos que sirvan para interactuar unos con otros. Todo lo estudiado en lecciones pasadas con respecto a tipos de variables, funciones y estructuras de control puede incluirse en las clases, con lo cual, se abre un amplio mundo de posibilidades con las que experimentar.

3.4. Variables de instancia vs variables de clase

Hablemos ahora sobre otro detalle que puede resultar bastante importante en las preguntas del examen, la diferencia que existe entre las variables de instancia y las variables de clase.

El nombre por el cual nos referimos a estos dos tipos de variables puede darnos pistas muy valiosas sobre la naturaleza de las mismas. La idea detrás de esto es que las "variables de instancia" tienen una relación muy estrecha con las instancias u objetos, mientras que las de clase, están ligadas a las clases.

Es posible que dos objetos de la misma clase tengan diferentes atributos, y no solo nos referimos al valor de los mismos, ya que contamos con ello y es una de las enormes ventajas de la POO, sino que también pueden añadirse más atributos a los objetos tan solo utilizando la nomenclatura del punto. Veamos un ejemplo.

```
1.      >>> class MiClase():
2.      >>>     def __init__(self, arg):
3.      >>>         self.propiedad_1 = arg
4.
5.      >>> mi_objeto = MiClase(3)
6.      >>> mi_objeto.propiedad_2 = "segunda propiedad"
7.      >>> print(mi_objeto.propiedad_1,
8.                mi_objeto.propiedad_2, sep = " || ")
9.      3 || segunda propiedad
10.
```

En el ejemplo anterior, vemos como se define el atributo “propiedad_1” dentro del constructor, con lo cual se creará dicho atributo en el momento de instanciar la clase. Sin embargo, en la línea 6 vemos cómo es posible también añadir otro atributo, “propiedad_2” al objeto existente.

Como es lógico, si creáramos un nuevo objeto instanciando la clase “miClase”, este nuevo objeto tendría un valor del atributo “propiedad_1” acorde al parámetro pasado en la construcción y además carecería del atributo “propiedad_2” añadido anteriormente al objeto “mi_objeto”.

Además, se pueden añadir atributos dentro de otro método de la clase. En ese caso sólo los objetos que llamen a ese método tendrán esos atributos.

```
1.      >>> class MiClase():
2.      >>>     def __init__(self, arg):
3.      >>>         self.propiedad_1 = arg
4.      >>>
5.      >>>     def metodo(self, arg):
6.      >>>        self.propiedad_2 = arg
7.
8.      >>> mi_objeto = MiClase(3)
9.      >>> print(mi_objeto.propiedad_1,
10.               mi_objeto.propiedad_2, sep = " || ")
11.     Traceback
12.     AttributeError: 'MiClase' object has no attribute '
13.     propiedad_2’
14.
15.     >>> mi_objeto.metodo("segunda propiedad")
16.     >>> print(mi_objeto.propiedad_1,
17.               mi_objeto.propiedad_2, sep = " || ")
18.     3 || segunda propiedad
19.
```

Puede sonar un tanto complejo, pero, en definitiva, la conclusión a la que se pretende llegar es que alterar este tipo de variables, llamadas **variables instancia** surtirá efecto dentro del objeto al que pertenecen, pero no afectará a otros objetos ni al valor de sus atributos.

Además, otra de las características fundamentales de las variables instancia es que solo existen si existe algún objeto, ya que estas se crean al construir el objeto o modificar uno existente, de otra manera no sería posible su existencia.

Ante esta posibilidad de que diferentes objetos posean diferentes variables instancia, debería existir una manera de asegurar la existencia de las mismas para evitar llamar alguna inexistente y por tanto causar errores. Si bien es cierto que podemos tratar esta situación utilizando las excepciones, Python también provee una manera de conocer los elementos de los que podemos disponer en un objeto.

Cada vez que se crea un objeto, Python crea a la vez una serie de propiedades y métodos predefinidos del mismo, no son opcionales, se crean sí o sí. Entre otras, cada vez que instanciamos un objeto, Python provee al mismo con una variable llamada "**__dict__**" que como seguramente se podrá imaginar, es un diccionario. Este diccionario contiene los nombres y valores de todos los atributos y todos los métodos que el objeto contiene. Veamos un ejemplo:

```
1.      >>> class MiClase():
2.      >>>     def __init__(self, arg):
3.      >>>         self.propiedad_1 = arg
4.
5.      >>> mi_objeto_1 = MiClase(3)
6.      >>> mi_objeto_1.propiedad_2 = "segunda propiedad"
7.      >>> mi_objeto_2 = miClase(100)
8.
9.      >>> print(mi_objeto_1.__dict__)
10.     {
11.        "propiedad_1": 3,
12.        "propiedad_2": "segunda propiedad"
13.     }
14.     >>> print(mi_objeto_2.__dict__)
15.     {
16.        "propiedad_1": 100
17.     }
18.
```

Por otro lado, podemos encontrarnos con las **variables de clase**. Estas variables, por el contrario, están mucho más ligadas a la clase que a los objetos. Observemos un ejemplo de variable de clase.

```
1.      >>> class ClaseEjemplo:
2.      >>>     contador = 0
3.      >>>     def __init__(self, val = 1):
4.      >>>         self.__primera = val
5.      >>>         ClaseEjemplo.contador += 1
6.
7.      >>> objetoEjemplo1 = ClaseEjemplo()
8.      >>> objetoEjemplo2 = ClaseEjemplo(2)
9.      >>> objetoEjemplo3 = ClaseEjemplo(4)
10.
11.     >>> print(objetoEjemplo1.contador,
12.                objetoEjemplo2.contador,
13.                objetoEjemplo3.contador, sep = " || ")
14.     3 || 3 || 3
15.
```

La manera de definir esta variable de clase es bastante intuitiva, no se hace dentro del constructor, ya que si se hiciera de esta manera estaríamos hablando de una variable de instancia. Podemos observar que al imprimir por pantalla el valor de la variable de clase “contador” (línea 11), el resultado es el mismo para todos los objetos.

Esta es la clave para entender la diferencia entre los dos tipos de variables. Una variable de clase está disponible para todos los objetos que se instancien, de hecho, existe sin necesidad de que se cree ningún objeto, al contrario que con las variables instancias.

En este ejemplo, esta variable de clase nos ayuda a saber cuántos objetos de esta clase han sido creados.

Finalmente, para concluir con este apartado, mencionaremos otra de las herramientas que tenemos disponibles en Python para conocer la existencia o inexistencia tanto de una variable de instancia como de una variable de clase.

Esta herramienta es la función **hasattr()**. Esta función recibe 2 parámetros de entrada. El primero es el nombre de la clase u objeto y el segundo el nombre de la propiedad cuya existencia se quiere comprobar dentro de la clase u objeto.

Esta función devuelve el valor booleano “True” si la propiedad existe o “False” si esta es inexistente.

```
1.      >>> class miClase():
2.      >>>     variable_clase = 10
3.      >>>     def __init__(self,a,b):
4.      >>>         self.a = a
5.      >>>         self.b = b
6.
7.      >>> obj = miClase(1,2)
8.
9.      >>> print(hasattr(obj,"a"),
10.                hasattr(obj,"b"),
11.                hasattr(obj,"c"),
12.                hasattr(miClase,"variable_clase"),
13.                sep = " || ")
14.     True || True || False || True
15.
16.     >>> print(hasattr(miClase,"a"),
17.                hasattr(miClase,"b"),
18.                hasattr(obj,"variable_clase"),
19.                sep = " || ")
20.     False || False || True
21.
```

Algunas consideraciones importantes con respecto a la utilización de la función “hasattr()”:

> El parámetro de entrada que representa el nombre de la propiedad a comprobar siempre debe ser un dato de tipo cadena, cualquier otra cosa fallará.

> Con esta función podemos resaltar la diferencia existente entre las variables de instancia y las variables de clase, ya que como vemos en el ejemplo, el objeto no contiene la propiedad de clase “variable_clase” y por eso devuelve “False”, de la misma manera, preguntar si la clase “miClase” tiene el atributo “a” o “b” devolverá un “False” ya que estos atributos pertenecen al objeto y no a la clase.

4. Herencia

La importancia del concepto de herencia en programación viene de que utilizarla nos permite crear clases, pero sin partir desde cero, es decir, nos permite reutilizar el trabajo a la vez que mantenerlo ordenado. Si necesitamos de una clase más específica que se centre sobre un subgrupo de objetos, aquí es donde la herencia resulta de gran ayuda.

Cuando decimos que las subclases heredan ciertas características de las superclases simplemente nos referimos a atributos y métodos que esta subclase tiene la capacidad de utilizar como si estuvieran definidos dentro de ella misma.

4.1. Herencia única

Este tipo de herencia, como puede deducirse por su nombre, hace referencia a que las subclases implicadas provienen directamente solo de una clase padre o superclase. El siguiente esquema ilustra claramente esta situación.

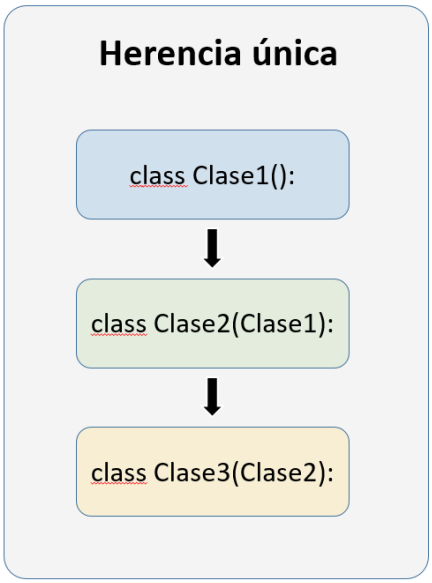

Figura 4.1. Esquema explicativo de la herencia única

4.1.1. Definición de una relación superclase-subclase en Python

Al igual que hicimos a la hora de definir una clase o un objeto, construyamos la relación de herencia más simple que podemos encontrar:

```
>>> class Superclase:
>>>     pass

>>> class Subclase(Superclase):
>>>     pass

```

Este ejemplo nos sirve para mostrar cual es la sintaxis para indicar que determinada clase es subclase de otra superclase. A la hora de definir la subclase, tan solo tenemos que escribir entre paréntesis el nombre de la superclase a la que pertenece.

4.1.2. Variables de clase en la herencia

El ejemplo anterior es demasiado sencillo, ya que las clases no tienen ni atributos ni métodos con los que "jugar" para mostrar las características más relevantes del trabajo con la herencia en Python.

Por ello, introduzcamos un nuevo ejemplo para ver primeramente cómo es posible que las subclases tengan acceso a las variables de clase de las superclases.

```
>>> class Superclase():
>>>     supervar = 5

>>> class Subclase(Superclase):
>>>     subvar = 1

>>> obj_sub = Subclase()
>>> print(obj_sub.supervar,
          obj_sub.subvar,sep = " || ")
5 || 1

```

Como podemos observar, las definiciones de las clases "Superclase" y "Subclase" siguen siendo muy simples, lo único que hemos añadido han sido las variables de clase "supervar" y "subvar".

Observamos que al instanciar el objeto como miembro de la clase “Subclase” e intentar imprimir por pantalla el valor de la variable de la superclase “supervar” no obtenemos ningún error.

De este ejemplo podemos extraer que, al definir una clase como subclase de otra, esta tiene automáticamente acceso a las variables de clase de la superclase de la que procede. Los objetos de esta subclase tendrán también esta capacidad, tal como hemos comprobado.

4.1.3. Herencia de métodos

Continuemos con el siguiente paso en orden de dificultad. La herencia de métodos. No solo es posible heredar variables, sino que también “funcionalidades” lo cual se traduce en métodos a los que la subclase podrá tener acceso por la relación que esta tiene con su superclase.

Continuemos con el ejemplo del apartado anterior, solo que añadiendo un método muy simple.

```
1.      >>> class Superclase():
2.      >>>     supervar = 5
3.      >>>     def superfunc(self):
4.      >>>         return "Hello I'm the superfunc"
5.
6.      >>> class Subclase(Superclase):
7.      >>>     subvar = 1
8.
9.      >>> obj_sub = Subclase()
10.     >>> print(obj_sub.supervar, obj_sub.subvar,
11.                obj_sub.superfunc(), sep = " || ")
12.     5 || 1 || Hello I'm the superfunc
13.
```

Podemos observar cómo hemos añadido un nuevo método de la superclase en las líneas 3 y 4. Este método lo único que hace es devolver la cadena “Hello I’m the superfunc” nada del otro mundo.

Nuevamente, podemos observar cómo al instanciar un objeto de la subclase, tenemos acceso al método de la superclase por el mero hecho de la relación existente entre superclase y subclase. Así es como funciona la herencia con respecto a los métodos.

Si nuestra clase es una subclase de otra superclase, heredará de esta última la capacidad de invocar sus métodos.

4.1.4. Variables de instancia en la herencia

Continuemos complicando un poco el ejemplo. A continuación, hablaremos de cómo es posible que las variables de instancia (aquellas que pertenecen a los objetos) también estén al alcance de la subclase y consecuentemente de los objetos que sean instancias de la misma.

```
>>> class Deportista:
>>>     def __init__(self,deporte):
>>>         self.deporte = deporte
>>>
>>> class Futbolista(Deportista):
>>>      def __init__(self, deporte, posicion,
                                        equipo):
>>>          Deportista.__init__(self,deporte)
>>>          self.posicion = posicion
>>>          self.equipo = equipo
>>>
>>> messi = Futbolista("fútbol", "delantero",
                        "barcelona")
>>> print(messi.deporte, messi.posicion,
                sep = " || ")
fútbol || delantero
>>>
```

En este segundo ejemplo, partimos de una superclase “Deportista” y creamos una subclase llamada “Futbolista”.

Con este ejemplo podemos entender de una manera mucho más clara la relación entre subclase y superclase. “Deportista” es la superclase porque engloba a una categoría más particular que es la de “Futbolista”.

También nos sirve para comenzar a ver de una manera práctica lo que implica la herencia entre superclases y subclases.

Podemos observar como en las primeras líneas del código se defina la superclase “Deportista” dentro de la misma invocamos al constructor e inicializamos el objeto para que al instanciarse tenga el atributo “deporte” el cual hace referencia al deporte dentro del cual el deportista desarrolla su

carrera. Hasta aquí nada nuevo, simplemente la definición de una clase en Python.

En la segunda parte del código (líneas 6-11) definimos la subclase con el nombre de “Futbolista”, indicando al definir la misma, que será una subclase de “Deportista”. La novedad la encontramos dentro del constructor de esta subclase, ya que en la línea 8 invocamos también el constructor de la superclase.

Al hacer esto, le estamos diciendo a Python que queremos que se ejecute la inicialización de la superclase también, lo cual permitirá que el objeto instanciado de la subclase, posea además de sus propios atributos, aquellos presentes en el constructor de la superclase.

Podemos comprobar que esto es así en las líneas 13 y 14, dónde imprimimos por pantalla los atributos “deporte” y “posición” del objeto de la subclase. Sin haber tenido que definir el atributo “deporte” dentro del constructor de la subclase, hemos sido capaces de invocarlo e imprimirlo.

La regla que podemos extraer es que, si queremos disponer de los atributos declarados en el constructor de la superclase, tenemos que invocar la función “__init__” de la superclase dentro del constructor de la subclase con la sintaxis que vemos en la línea 8:

NombreSuperClase.__init__(self,arg1,arg2,..,argN)

Aun así, cabe destacar que, habitualmente se suele hacer uso de la función “super()” para este fin, ya que al invocarla, automáticamente Python entiende que estamos invocando a la clase padre de la subclase. A continuación, el mismo ejemplo anterior solo que haciendo uso de dicha función.

```
>>> class Deportista:
>>>     def __init__(self,deporte):
>>>         self.deporte = deporte
>>>
>>> class Futbolista(Deportista):
>>>         def __init__(self, deporte, posicion,
                                        equipo):
>>>             super().__init__(deporte)
>>>             self.posicion = posicion
>>>              self.equipo = equipo
>>>
>>> messi = Futbolista("fútbol", "delantero",
                                        "barcelona")
>>> print(messi.deporte, messi.posicion,
                                        sep = " || ")
fútbol || delantero
>>>
```

Como vemos, al utilizar la función “super()” no es necesario pasar el argumento “self” dentro del constructor, ya que la misma función se encarga de apuntar al mismo objeto que se está instanciando, quitando la necesidad de indicarlo.

Como veremos más adelante, esta función no solo se utiliza para invocar al constructor de la superclase, sino que también para acceder a cualquiera de los recursos disponibles dentro de la misma. Comenzamos aquí a ver el gran potencial que tiene la utilización de la herencia.

Para terminar con este apartado de **herencia única**, introduzcamos un nivel más de herencia, es decir una tercera clase y saquemos conclusiones sobre su comportamiento.

```
>>> class Clase1():
>>>     varclase1 = "variable clase 1"
>>>     def __init__(self, var1):
>>>         self.var1 = var1
>>>     def func1(self):
>>>         return "I'm func1"

>>> class Clase2(Clase1):
>>>     varclase2 = "variable clase 2"
>>>     def __init__(self, var1, var2):
>>>         super().__init__(var1)
>>>         self.var2 = var2
>>>     def func2(self):
>>>         return "I'm func2"

>>> class Clase3(Clase2):
>>>     varclase3 = "variable clase 3"
>>>     def __init__(self, var1, var2, var3):
>>>         super().__init__(var1,var2)
>>>         self.var3 = var3
>>>     def func3(self):
>>>         return "I'm func3"

>>> obj_clase3 = Clase3(1,2,3) #Instanciación de la
                               # clase

>>> print(obj_clase3.__dict__, obj_clase3.func1(),
          obj_clase3.func2(), obj_clase3.func3(),
          sep = " || ")
{'var1': 1, 'var2': 2, 'var3': 3} || I'm func1 ||
 I'm func2 || I'm func3

>>> print(obj_clase3.varclase1, obj_clase3.varclase2,
          obj_clase3.varclase3, sep = " || ")

variable clase 1 || variable clase 2 || variable
clase 3

```

Utilizaremos este ejemplo para resumir las principales características del comportamiento de las clases cuando se relacionan mediante una herencia única.

Esta herencia única significa que ninguna clase tiene más de una superclase de la que proceda, lo cual se traduce a efectos de código en que, en la definición de la clase, no aparezcan entre paréntesis varios nombres de clases. Estudiaremos esta situación, un poco más compleja, en el siguiente apartado de la lección.

Volviendo al ejemplo, este nos será muy útil para resumir las reglas que sigue Python para trabajar con la herencia.

Observamos como en este caso hemos definido 3 clases: “Clase1”, “Clase2” y “Clase3”, dónde “Clase1” es la superclase principal, la clase “Clase2” es subclase de la “Clase1” y finalmente, “Clase3” es subclase de “Clase2”, lo cual significa que también es subclase de “Clase1” ya que esta se encuentra en el primer nivel de herencia. Gráficamente podemos verlo como sigue

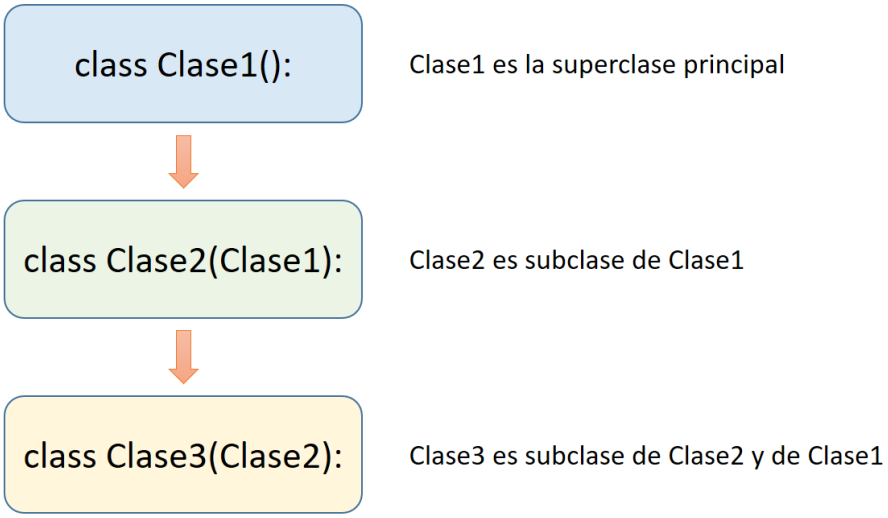

Figura 2.3. Esquema de la relación de herencia única entre clases

Dentro de cada una de estas clases hemos definido respectivas variables de clase (varclase1, varclase2 y varclase3). También cada una de estas clases tiene un constructor dentro del cual se utiliza la función “super()” para invocar el constructor de la relativa superclase. Finalmente, dentro de cada clase hemos definido respectivos métodos: func1, func2 y func3, cuya principal finalidad es devolver las cadenas de texto que pueden observarse en las líneas 6, 14 y 22 cuando son invocados.

En resumidas cuentas, hemos dotado a cada una de las clases de todos los tipos de “recursos” que pueden tener para comprobar cuál es el comportamiento cuando se intenta acceder a cada uno de estos recursos desde la subclase más baja, es decir, la “Clase3”.

Para esto, primeramente, hemos instanciado la “Clase3” creando el objeto obj_clase3 introduciendo los parámetros de entrada necesarios para realizar esta definición.

Posteriormente, hemos hecho un primer print donde llamamos a la variable "__dict__" (recordemos que esta variable era creada por Python de manera automática cada vez que creamos un objeto) la cual contiene los atributos de los que dispone el objeto.

Vemos claramente como no hay ningún tipo de problema en acceder a los atributos "var1" y "var2" debido a la relación que tiene la clase 3 con sus superclases "Clase2" y "Clase1".

De la misma manera, intentamos invocar las funciones "func1" y "func2" y comprobamos como tenemos pleno acceso a estas funciones. Finalmente, realizamos la misma comprobación con las variables de clase "varclase1" y "varclase2", las cuales imprimimos por pantalla sin que Python arroje ningún tipo de error.

El resumen de la manera que tiene de actuar Python con este tipo de herencia puede resumirse en los siguientes puntos:

- La subclase tiene acceso a las variables de clase y a los métodos de la superclase simplemente por la relación de herencia que tiene con ella.

- En el caso de los atributos que podemos encontrar en el constructor de las clases, si queremos que la subclase tenga acceso a ellos, tenemos que invocar al constructor de la superclase dentro del constructor de la subclase. Esto se puede hacer de manera explícita, es decir, a través del propio nombre de la superclase o bien utilizando la función "super()" tal y como hemos visto en los ejemplos.

4.1.5. Conflictos en herencia única

Hasta ahora, en todos los ejemplos que hemos utilizado no ha surgido ningún conflicto entre variables de clase, de instancia o métodos.

¿Cuándo se darán estos conflictos? Pues bien, estos conflictos pueden aparecer cuando tenemos variables con los mismos nombres tanto en la propia subclase como en las superclases de la misma.

Veamos un ejemplo muy sencillo de este tipo de casuística:

```
>>> class Superclase():
>>>     def __init__(self):
>>>         self.variable = 1
>>>
>>> class Sublase(Superclase):
>>>     def __init__(self):
>>>         super().__init__()
>>>         self.variable = 0
>>>
>>> obj_subclase = Sublase()
>>> print("El valor de la variable 'variable' es: ",
          obj_subclase.variable)
El valor de la variable 'variable' es:  0
```

En el ejemplo anterior, hemos definido dos clases, “Superclase” y “Subclase”. Dentro del constructor de cada una de ellas hemos definido el atributo “variable” (con el mismo nombre) y a cada uno de ellos le hemos asignado un valor según la clase. Posteriormente, hemos instanciado un objeto de la subclase y hemos hecho un print del atributo “variable” para comprobar cuál es el valor que tendrá este. Como podemos comprobar, el valor de “variable” se corresponde con el definido dentro de la subclase. Podemos comprobar que este comportamiento es extensible al caso de las variables de clase y los métodos igualmente, lo vemos en las siguientes líneas, donde se han resaltado las variables y métodos añadidos.

```
>>> class Superclase():
>>>     variable_clase = True
>>>     def __init__(self):
>>>         self.variable = 1
>>>     def fun(self):
>>>         return "Hello, I'm came from Superclase"
>>>
>>> class Subclase(Superclase):
>>>     variable_clase = "variable de subclase"
>>>     def __init__(self):
>>>         super().__init__()
>>>         self.variable = 0
>>>     def fun(self):
>>>         return "Hello, I'm came from Subclase"
>>>
>>> obj_subclase = Subclase()
>>> print(obj_subclase.variable_clase,
          obj_subclase.fun(), sep = "\n")
variable de subclase
Hello, I'm came from Subclase
```

Python intenta primeramente buscar la variable o método dentro del mismo objeto y subclase de la que es instancia, si no lo encuentra busca en las superclases en orden ascendente, lo cual significa que tomará el primer valor que se encuentre en la línea de herencia y parará cuando encuentre dicha coincidencia. Si no los encuentra en ninguna de las superclases arroja un error de tipo "AttributeError".

Como vemos, la lógica que sigue Python para resolver estos posibles conflictos es bastante lógica, valga la redundancia. En el siguiente apartado veremos cómo Python resuelve este tipo de conflictos cuando la herencia deja de ser única y pasa a ser **múltiple**.

4.2. Herencia múltiple

En los apartados anteriores hemos estudiado la manera en la que Python se comporta y resuelve conflictos cuando la herencia con la que trabajamos es única. En este apartado complicaremos un poco la situación y trataremos de extraer las reglas que sigue Python para trabajar con las herencias múltiples. Es importante aprender y entender cuáles son estas reglas, ya que las preguntas relacionadas que podemos encontrar en el examen suelen incidir sobre este aspecto de la relación entre clases y subclases.

Veamos inicialmente un esquema comparativo entre herencia única y herencia múltiple.

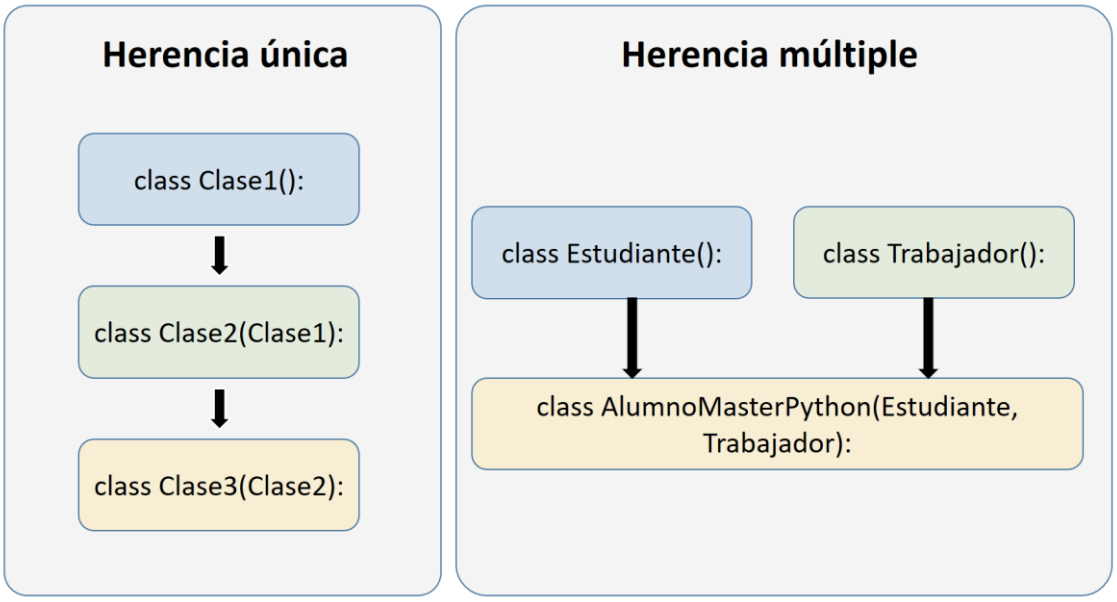

Figura 4.2. Esquema comparativo entre la herencia única y la herencia múltiple

Mediante la figura anterior podemos ver fácilmente cual es la diferencia entre los dos tipos de herencia.

La herencia múltiple se da cuando nos interesa crear una subclase que herede características de dos superclases que se encuentran "al mismo nivel". Como vemos mediante este ejemplo particular con el que posiblemente más de uno de los estudiantes de esta asignatura esté familiarizado, no es nada difícil ni extraño encontrar situaciones en las que sea necesario la utilización de la herencia múltiple.

Por ello, a continuación, iremos explicando cómo se define la herencia múltiple y el comportamiento que tiene Python ante estas. Sobre todo, nos interesa la cuestión de los conflictos que puedan surgir entre variables declaradas en ambas superclases, ¿qué valor tomarán las variables en este caso?, ¿qué reglas sigue Python para asignar estos valores? Lo vemos a continuación.

4.2.1. Definición de una clase con herencia múltiple

Comencemos con lo más simple, la definición de una clase con herencia múltiple. Para ello necesitamos obviamente primero de la existencia de mínimo 2 clases que serán las superclases de la subclase que queremos crear. Veamos en el siguiente ejemplo como codificar esta situación en Python.

```
1.      >>> class Super1():
2.      >>>     pass
3.
4.      >>> class Super2():
5.      >>>     pass
6.
7.      >>> class Subclass(Super1,Super2):
8.      >>>     pass
9.
10.     >>> print(issubclass(Subclass, Super1),
11.                issubclass(Subclass, Super2))
12.     True True
13.
```

En el ejemplo anterior creamos en primer lugar las dos superclases, "Super1" y "Super2" y posteriormente, en la línea 5 creamos la subclase "Subclass". Vemos como para indicar que la subclase proviene de varias superclases no

tenemos más que añadir sus nombres entre los paréntesis de la definición de la misma subclase.

Para demostrar que tan solo haciendo esto Python ya entiende que "Subclass" heredará características de las superclases hacemos uso de la función "issubclass(*Clase2,Clase1*)", la cual devuelve el valor booleano "True" si *"Clase2"* es subclase de *"Clase1" y "False"* en caso contrario. Comprobamos este hecho con los valores obtenidos en la línea 9.

4.2.2. Herencia múltiple. Variables de clase, atributos y métodos sin conflicto

La utilización de la herencia múltiple a la hora de construir subclases puede resultar problemática en algunos casos que veremos posteriormente, sin embargo, puede otorgar una gran cantidad de ventajas desde el punto de vista del desarrollo, ya que posibilita el acceso de los objetos de las subclases a todos los recursos disponibles de sus superclases.

Es decir, haciendo uso de la herencia múltiple, la subclase tendrá la capacidad de disponer de los métodos, atributos y variables de clase de sus superclases. Veamos un ejemplo de este tipo de situación.

```
1.      >>> class Clase1():
2.      >>>     varclase1 = "variable clase 1"
3.      >>>     def __init__(self, var1):
4.      >>>         self.var1 = var1
5.      >>>     def func1(self):
6.      >>>         return "I'm func1"
7.
8.      >>> class Clase2():
9.      >>>     varclase2 = "variable clase 2"
10.     >>>     def __init__(self,var2):
11.     >>>         self.var2 = var2
12.     >>>     def func2(self):
13.     >>>         return "I'm func2"
14.
15.     >>> class Clase3(Clase1, Clase2):
16.     >>>     varclase3 = "variable clase 3"
17.     >>>     def __init__(self,var1,var2,var3):
18.     >>>         Clase1.__init__(self,var1)
19.     >>>         Clase2.__init__(self,var2)
20.     >>>         self.var3 = var3
21.     >>>     def func3(self):
22.     >>>         return "I'm func3"
23.
24.     >>> obj_clase3 = Clase3(1,2,3) #Instanciación de la
25.                                     # clase
26.
```

```
>>> print("Las variables de clase son: ",
          obj_clase3.__dict__)
Las variables de instancia son:
{'var1': 1, 'var2': 2, 'var3': 3}

>>> print("Los atributos del objeto son: ")
>>> print(obj_clase3.var1, obj_clase3.var2,
          obj_clase3.var3, sep = " || ")
Los atributos del objeto son: 1 || 2 || 3

>>> print(obj_clase3.func1(), obj_clase3.func2(),
          obj_clase3.func3(), sep = " || ")
 I'm func1 || I'm func2 || I'm func3

```

El ejemplo anterior pretende ser un ejemplo de herencia múltiple sin ningún tipo de conflicto entre variables o métodos. No hay nombres de variables o métodos repetidos en las clases, solamente se ha hecho el ejemplo homólogo al de la herencia única sin conflictos. Hemos dotado a todas las clases de todos los recursos de los que pueden disponer para observar el comportamiento de los mismos al establecer entre la subclase y las superclases una herencia múltiple.

Hemos definido 3 clases: “Clase1”, “Clase2” y “Clase3”. Hemos dotado a cada una de ellas de una variable de clase con distintos nombres, respectivamente: “varclase1”, “varclase2” y “varclase3”. Y cada clase tiene atributos definidos mediante el constructor y un método todos ellos también con nombres distintos.

Observando el código y el comportamiento del mismo, podemos extraer una serie de conclusiones sobre el funcionamiento de la herencia múltiple en Python **cuando no existen conflictos**.

- La manera de declarar una herencia múltiple en una subclase es introduciendo los nombres de las superclases entre paréntesis en la definición de la subclase

  class Subclase(Superclase1,Superclase2,...,SuperclaseN):

- Al declarar esta herencia múltiple, la subclase (y los objetos de la misma) tiene directamente acceso a las variables de clase y métodos de las superclases.

> Si se quiere que la subclase tenga acceso a los atributos definidos mediante los constructores de las superclases es obligatorio invocarlas explícitamente tal y como se observa en las líneas 18 y 19. Esta es una de las principales diferencias con respecto a la situación homóloga de la herencia única, ya que, en dicho caso, solo es necesario invocar a la siguiente superclase en la línea de la herencia y no todas las superiores.

En líneas generales podemos decir que cuando no hay conflictos por nombres de variables o métodos repetidos en varias clases, las subclases con herencia múltiple se comportan de manera casi idéntica a el caso de herencia única, con la excepción de que en la herencia múltiple es necesario invocar explícitamente los constructores de las superclases.

4.2.3. Herencia múltiple. Resolución de conflictos

Ahora bien, ¿Qué ocurre cuando tenemos una situación de herencia múltiple y se genera algún tipo de conflicto entre las variables o métodos que tiene cada una de las clases? ¿Qué regla sigue Python para resolver estos conflictos y asignar los valores a las variables que entran en juego en las relaciones entre las clases y los objetos de las mismas?

Pues bien, nada mejor que algunos ejemplos para observar el comportamiento de Python y extraer las reglas que tenemos que aplicar para predecir el comportamiento de las clases que generemos en este tipo de situaciones. Comencemos por una situación sencilla.

```
>>> class Clase1():
>>>     varclase = "variable clase 1"
>>>     def func(self):
>>>         return "I came from Clase1"

>>> class Clase2():
>>>     varclase = "variable clase 2"
>>>     def func(self):
>>>         return "I came from Clase2"

>>> class Clase3(Clase1, Clase2):
>>>     varclase = "variable clase 3"
>>>     def func(self):
>>>         return "I came from Clase3"

>>> obj_clase3 = Clase3()
>>> print(obj_clase3.varclase, obj_clase3.func(),
          sep = " || ")
variable clase 3 || I came from Clase3
```

Podemos observar como en el ejemplo anterior hemos definido tanto en las dos superclases como en la subclase la variable de clase “varclase” y el método “func”. Al hacer el print de estas propiedades observamos cómo Python toma los valores definidos dentro de la propia subclase, lo cual significa que se sigue respetando la regla de que se comienza buscando primeramente dentro del mismo objeto para ver si se encuentran los atributos y posteriormente pasa a las superclases.

Repitamos el ejemplo, pero ahora omitiendo estos valores dentro de la propia subclase y observemos los valores que asigna Python a estas mismas variables.

```
>>> class Clase1():
>>>     varclase = "variable clase 1"
>>>     def func(self):
>>>         return "I came from Clase1"

>>> class Clase2():
>>>     varclase = "variable clase 2"
>>>     def func(self):
>>>         return "I came from Clase2"

>>> class Clase3(Clase1, Clase2):
>>>     varclase3 = "variable clase 3"
>>>     def func3(self):
>>>         return "I came from Clase3"

>>> obj_clase3 = Clase3()
>>> print(obj_clase3.varclase, obj_clase3.func(),
        sep = " || ")
variable clase 1 || I came from Clase1

```

Podemos comprobar cómo hemos realizado dos simples modificaciones, hemos añadido un "3" a la variable de clase y al método definidos en la subclase para que al hacer el print Python no encuentre estas variables dentro del mismo objeto y tenga que recurrir a la herencia para asignar los valores adecuados.

Observamos que, al realizar estas modificaciones, los valores obtenidos corresponden a la superclase "Clase1", de lo cual se puede extraer que cuando tenemos conflictos por tener mismo nombre de variables en las superclases, Python toma el valor de aquella superclase que se encuentra primero (a la izquierda) en la definición de la subclase. Es decir, si la definición de la subclase es la siguiente:

class Clase3(Clase1,Clase2):

Y tenemos conflictos por tener los mismos nombres en 2 o más superclases, Python escogerá el valor de aquella clase que se encuentre más a la izquierda, en este caso: "Clase1".

Con lo cual extraemos como regla general que **Python a la hora de resolver el valor de las variables de una subclase busca de abajo hacia arriba en la línea de herencia y de izquierda a derecha cuando hay varias superclases al mismo nivel.**

Es sumamente importante que memoricemos este comportamiento, ya que como hemos mencionado anteriormente, son frecuentes las preguntas en las que se tiene que poner en práctica esta regla en el examen PCAP.

4.3. Conclusión sobre la herencia

Hemos visto que la herencia puede aportar un gran número de ventajas a la hora de plantear la resolución de diferentes problemas haciendo uso de todas sus características, sin embargo, con respecto a la **herencia múltiple** es recomendable intentar evitarla siempre que sea posible, ya que como hemos visto pueden surgir conflictos que hagan que nuestro código se comporte de una manera impredecible y no termine cumpliendo su función finalmente.

5. Puntos Clave

Los siguientes puntos resumen el contenido visto en esta lección.

- El paradigma de la programación orientada a objetos (POO) es sumamente relevante, tanto en el mundo profesional como en el examen final del PCAP, ya que representa aproximadamente un tercio del contenido total.
- Hemos explicado brevemente la idea conceptual detrás de este paradigma, mencionando los aspectos más importantes del mismo. Estos son: clases, objetos y herencia.
- Hemos visto la manera de codificar las ideas de la POO en Python, cuáles son las palabras reservadas, cómo definir una clase y como instanciar un objeto.
- Hemos visto la manera en la que se inicializan las propiedades de los objetos en Python mediante la función __init__ o constructor.
- Hemos visto la manera en la que se pueden definir los atributos de los objetos.
- Así mismo, también hemos construido métodos que permitan realizar acciones a estos métodos.
- Hemos hecho énfasis en la importancia de entender la diferencia entre variables de instancia y variables de clase y hemos visto algunas herramientas para poder distinguirlas y saber si existen.
- Hemos visto la manera en la que la relación de herencia se define en Python. Viendo que la herencia en Python puede dividirse en dos categorías: herencia única y herencia múltiple.
- Hemos estudiado de qué manera las subclases heredan los diferentes “recursos” (variables de clase, atributos y métodos) de las superclases. Tanto en la situación de herencia única como en la de herencia múltiple.

Tanto en herencia única como múltiple, hemos visto que es posible que haya conflictos entre clases y que pueden darse cuando dentro de ellas tenemos variables y/o métodos con el mismo nombre. Para resolver estos conflictos, hemos observado el método de resolución que sigue Python, el cual puede resumirse como sigue:

Python busca primeramente las variables dentro del objeto mismo, luego en orden ascendente en la jerarquía de clases y de izquierda a derecha en el caso de que ocurriera una herencia múltiple, parándose cuando encuentra la primera coincidencia.